TCS — Blockchain Tutorial 2

# Explore Blockchain with a Simple JavaScript Example

To reinforce your understanding of the role of blockchain we illustrate the main concepts by building a simple blockchain using JavaScript. This tutorial is adapted from a YouTube video that very nicely illustrates how the blockchain works:

1. Part 1 — [Create a Blockchain in JavaScript](https://www.youtube.com/watch?v=zVqczFZr124)

* This part of the tutorial implements a very basic blockchain structure in JavaScript, which in no way could be regarded as usable, it merely illustrates concepts.

1. Part 2 — [Implementing a ‘Proof of Work’ in JavaScript](https://www.youtube.com/watch?v=HneatE69814)

* This part of the tutorial adds a proof of work method to increase the difficulty of calculating the validity of the block, again merely to illustrates concepts

1. Part 3 — Rebuild your JavaScript blockchain example in PHP

* This part of the tutorial is optional exercise asking you to convert the JavaScript blockchain to utilise the SQLIte database. You will need:
  + A new SQLIte PDO database connection
  + PHP PDO model classes and associated methods to reflect the JavaScript Classes and methods
  + PHP controller methods to reflect the JavaScript methods in Part 1 and 2
  + Use JavaScript and JQuery to represent the miner process and invoke the new controller methods to append validated blocks to the new blockchain (based on SQLite)
  + Process any returned messages via a JSON object, e.g. false block, true block, etc.
  + …

You may simply click on the links above to follow the tutorials in YouTube or follow the brief written tutorials Part 1 and Part 2 below, Part 3 is down to you to investigate.

**Side note:**

You should note that there are many other elements of a blockchain’s block structure missing in this simple tutorial. For example, [BLOCKCYPHER](https://www.blockcypher.com/dev/bitcoin/#introduction) allows you to explore a number of blockchains including: Bitcoin, Ethereum, Litecoin, Dogecoin, and Bitcoin Testnet3

BlockCypher is a simple, mostly RESTful JSON API for interacting with blockchains, accessed over HTTP or HTTPS from the [api.blockcypher.com](https://api.blockcypher.com/v1/btc/main) domain.

A typical block representing the current state of a particular block that might be returned for a BlockHash or BlockHeight API endpoint will have many attributes such as: Hash, height, depth, chain, total, … a full list of bitcoin block attributes and their description can be found on the BLOCKCYPHER [bitcoin block description](https://www.blockcypher.com/dev/bitcoin/#block).

As an example, if you want to examine any particular bitcoin transaction based on its blockheight, assuming you know the block height you can run this API endpoint:

<https://api.blockcypher.com/v1/btc/main/blocks/496500?txstart=1&limit=1>

This happens to be, at the time of writing, the latest blockheight. Figure 1 shows the returned JSON formatted block.



Figure 1: Bitcoin latest block at time of writing, see timestamp (time).

Further, the technology behind blockchain uses advanced cryptography protocols, peer to peer networks, and many other technological solutions. So, it is highly unlikely that you would develop a complete new blockchain technology each time we need a blockchain for a new project. Instead you would most likely build upon existing open source blockchain technologies such as [Ethereum](https://www.ethereum.org/) or [Hyperledger](https://hyperledger.org/). Or, in terms of the research on visualising blockchain currency transactions outlined in the presentation we would use published API endpoints to develop our application.

If you want some quick background reading on blockchain, here is a three-part series from marmalab on their blog page that looks interesting:

* [The Blockchain Explained to Web Developers, Part 1: The Theory](https://marmelab.com/blog/2016/04/28/blockchain-for-web-developers-the-theory.html)
* The Blockchain Explained to Web Developers, Part 2: In Practice
* The Blockchain Explained to Web Developers, Part 3: The Truth

Note, these articles were posted in 2016, things change, new opinions are formed, technology moves on!

Ok, let’s get started with Part 1.

## Part 1 — Create a Blockchain in JavaScript

Our initial presentation gave you a feel for what a blockchain is, how it is structured, how it works. Part 1 of this Blockchain Tutorial 2 is based on the [YouTube video](https://www.youtube.com/watch?v=zVqczFZr124) indicated above, and has been selected because it very neatly illustrates some important blockchain concepts, while also allowing you to reinforce these concepts through code experimentation.

However, we should be aware that several underlying concepts are missing. So, this blockchain would not work very well in practice.

In this Blockchain Tutorial 2, Part 1, we will call our blockchain 'myBitcoin' and we will use OOP (Object Oriented Programming) for JavaScript. You will recall we used OOP for PHP to create our Model in our Blockchain Tutorial 1.

### Create a block structure

So, step by step:

1. First, we should create a JavaScript file in our MVC frameworks scripts folder. Call this JavaScript file jsBlockchain.js. Add, a comment of some sort to the top to describe what the JavaScript file is going to do, good practice.
2. Next, let’s create a block class. We start by creating a constructor class called 'block' to represent a block in the blockchain. This block class will hold methods to represent the main components of the block, such as: the block hash, previous hash, data, timestamp, and so on, Figure 2. Check out what attributes a typical bitcoin block structure contains on the [BlockCypher block page](https://www.blockcypher.com/dev/bitcoin/#block), to realise this is indeed a very simple structure.



Figure 2: Create the Class Block and its constructor.

* Code explanation
  + Line 2-9: This is the Block class with its constructor
  + Line 5-7: This is the constructor of the class Block that holds the block properties: index, timestamp, data, and previoushash. From line 6 we will add some methods.
    - 'index' is where the block sits in the blockchain.
    - 'timestamp' tells us when the block was created.
    - 'data' holds any type of data you want to hold in the block, e.g. bitcoins transacted, address of sender, address of receiver, etc.
    - 'previoushash' is the hash value of the previous block in the blockchain, which insures the integrity of the blockchain, and is initially set to empty, as we have no previoushash in the genesis block.

1. We now need to keep track of the block properties: index, timestamp, data and previoushash using the JavaScript keyword this to own the Block class constructor object properties (I think), Figure 3.

* Code explanation:
  + 7 – 10: The value of ‘this’ in JavaScript, will become the new object when the constructor is used to create an object. Check this out: [Gentle explanation of ‘this’ keyword in JavaScript](https://dmitripavlutin.com/gentle-explanation-of-this-in-javascript/). Also, read this [[quir]{mode]](https://www.quirksmode.org/js/this.html) for a good explanation.
    - The upshot of this is that when you invoke the construction (with its class Block properties) this.index = index; this.timestamp = timestamp, etc. become global variables (I think).
  + Line 12: We also have to calculate the current block hash, so we will need a global variable for this, which at the moment is set to empty

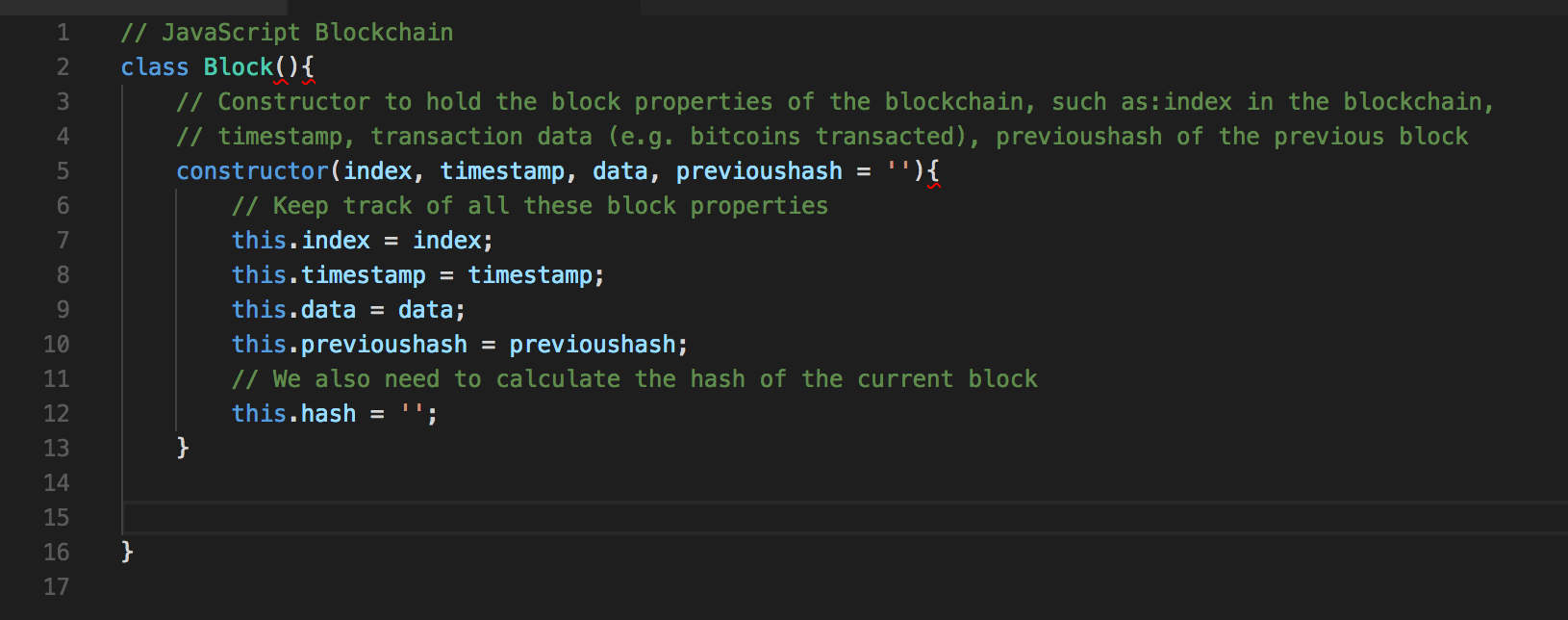


Figure 3: Keep track of the block properties

1. We need to create a method to calculate the current block hash using a JavaScript hash library such as SHA256. This method in the constructor will take the current block properties index, timestamp, data, previoushash and run them through a hash function to create the current block hash used to identfy this block on the blockchain.   
     
   Now, JavaScript does not have a native hash library so we need to import a hash library, e.g. crypto.js. I am using Visual Studio Code for my developer environment, because it has lots of nice features such as: an integrated terminal window, integrated GitHub, and so on. So, you will need to access a terminal window to install crypto.js.   
     
   When you have opened up a terminal window you should check that you have node.js installed, which is an open source server framework. There are many online resources that show you how to install node.js. In my case, I just checked it was already install by typing node -v in my terminal window, which returned v6.10.0, which is the version I have already installed.   
     
   This means that NPM, the node package manage, is already installed, so we can use this to install crypto.js. Cool!  
     
   You will need to type ‘npm install --save crypto-js’ on the terminal command line. You will also need to decide where to install the SHA256 library and as these are JavaScript libraries, perhaps you should change directory in your terminal window to your scripts folder and install there  
     
   Go ahead and type ‘npm install --save crypto-js’ on the terminal command line. This will install the node\_modules folder containing the crypto-js folder contacting in turn a load of JavaScript files including the sha256.js file needed to calculate the current block hash. Figure 4 shows the installation on the terminal window in Visual Studio Code.



Figure 4: Install the node\_modules/crypto-js/sha256.js files used to calculate the current block hash value

1. Now we can start to create the calculate hash method and use the SHA256 library to do this, Figure 5.

* Code explanation
  + Line 2: This now includes (requires) the SHA256 JavaScript hash library, which is used to calculate the hash value of the current block.
  + Line 19 – 23: This is the calculateHash() method in the Block class, which uses the SHA256 JavaScript hash library to calculate the hash. We input to the SHA256 function the current index, previoushash, timestamp and data. In doing so we convert the JavaScript data into JSON string format, and also cast the hash result into a string format, otherwise SHA256 returns an object.
  + Line 15: We now modify this.hash to calculate the current block hash value.

## ../../../../Desktop/Screen%20Shot%202017-11-23%20at%2015.37.05.png

Figure 5: Add the calculatehash() method

Next, we should create a blockchain class with a constructor that initialises the blockchain.

### Create a blockchain structure

1. Create a new class for our blockchain, Figure 6.

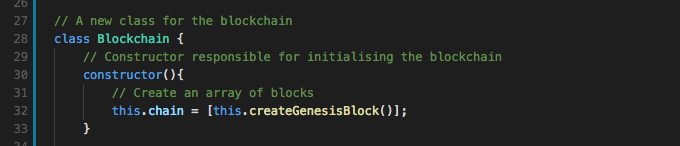


Figure 6: Create the new blockchain class to chain the blocks together

* Code explanation
  + Line 19 – 23: Clearly, we are creating a new blockchain class to chain the blocks together.
  + Line 30 – 32: Is a constructor method responsible for initialising the blockchain with an array of blocks.
  + Line 32: Creates a chain with an array of blocks.

1. Next, we need to initialise the blockchain with a genesis block, the first block in the blockchain, Figure 7.

* Code explanation
  + Line 36 -39: Create a method to create the genesis block, where
  + Line 38 returns the new (Genesis) block with appropriate values.

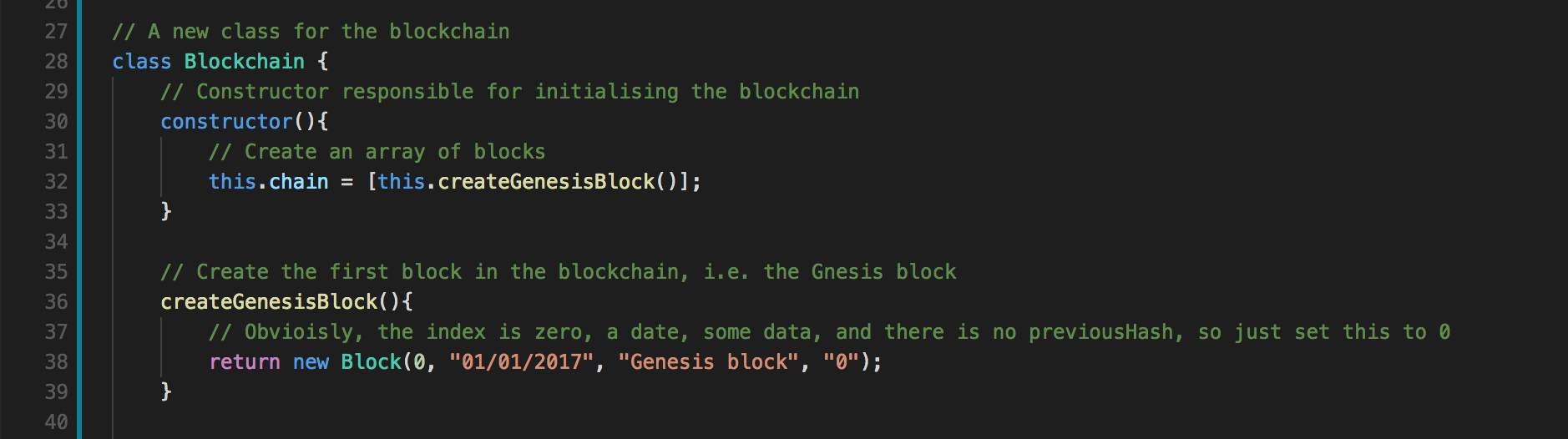


Figure 7: Create the genesis block

1. Next, we will need a couple of other methods, for example: the ability to get the latest block and add a new block to the blockchain, Figure 8

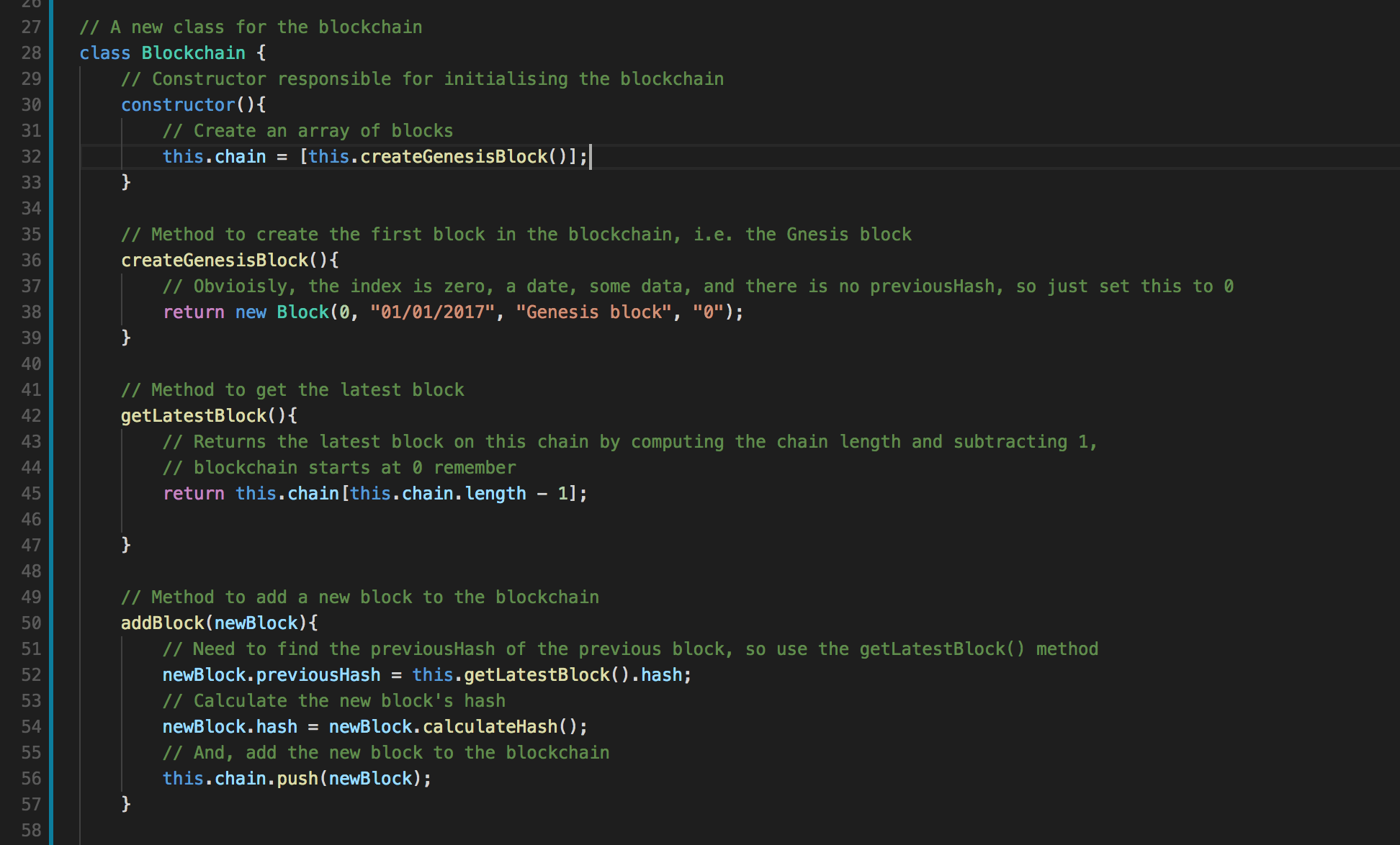


Figure 8: Create the getLatestBlock() and addBlock() methods.

* Line 42 – 47: The methods to get the latest block, where
* Line 45 returns the latest block index by finding the chain length and subtracting 1.
* Line 50 – 58: Create the addBlock method, where
* Line 52: Gets the latest block has to create the previousHash values to append to this new block
* Line 54: Creates the new block hash using the SHA256 based calculateHash() method, and
* Line 56 adds the new block to the blockchain.

1. Next, we need to test if all this works so far, so we will use a terminal window to see our results. But, first we need to create the new blockchain, add a couple of new blocks to the blockchain and use conole.log to view the results in the terminal window, Figure 9.

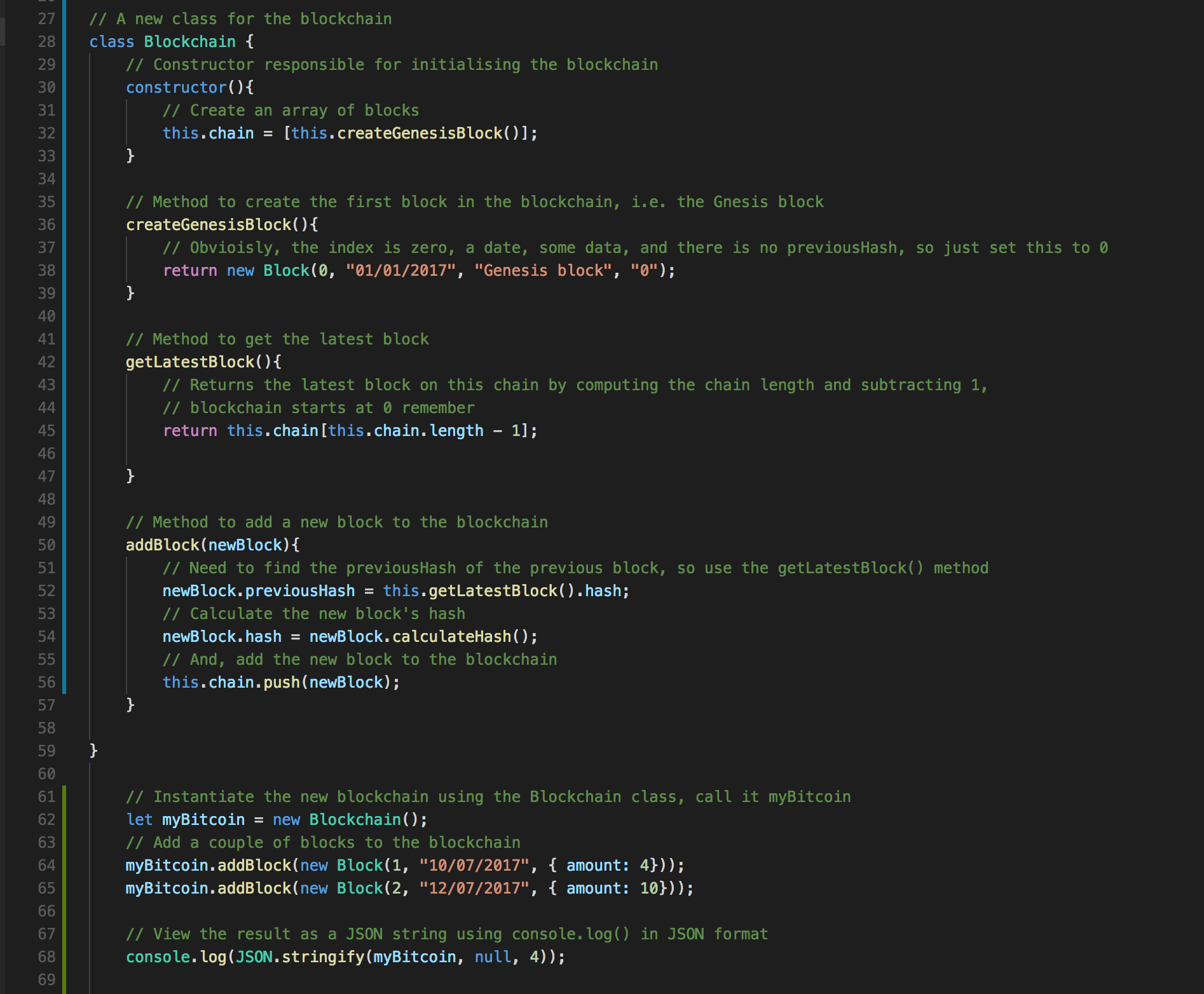


Figure 9: Test the blockchain

* Code explanation
  + Line 61: Instantiates the new blockchain (myBitcoin)
  + Lines 63 -64: Adds two new blocks to the blockchain
  + Line 67: Uses console.log to view the blockchian

1. You will need to open up a terminal window to test your blockchain by running your jsblockchain.js file, don’t forget to save it, debug it, etc. You should have node.js installed. Run node jsblockchain.js on the command line, Figure 10. We can see that we have output the blockchain, which has the property (or name) chain, which is also an array of blocks each containing an index, timestamp, data, previousHash, and hash (of the current block). If your does not work, then debug it before you move on. However, you can see that each block points back to the previous block via the previousHash

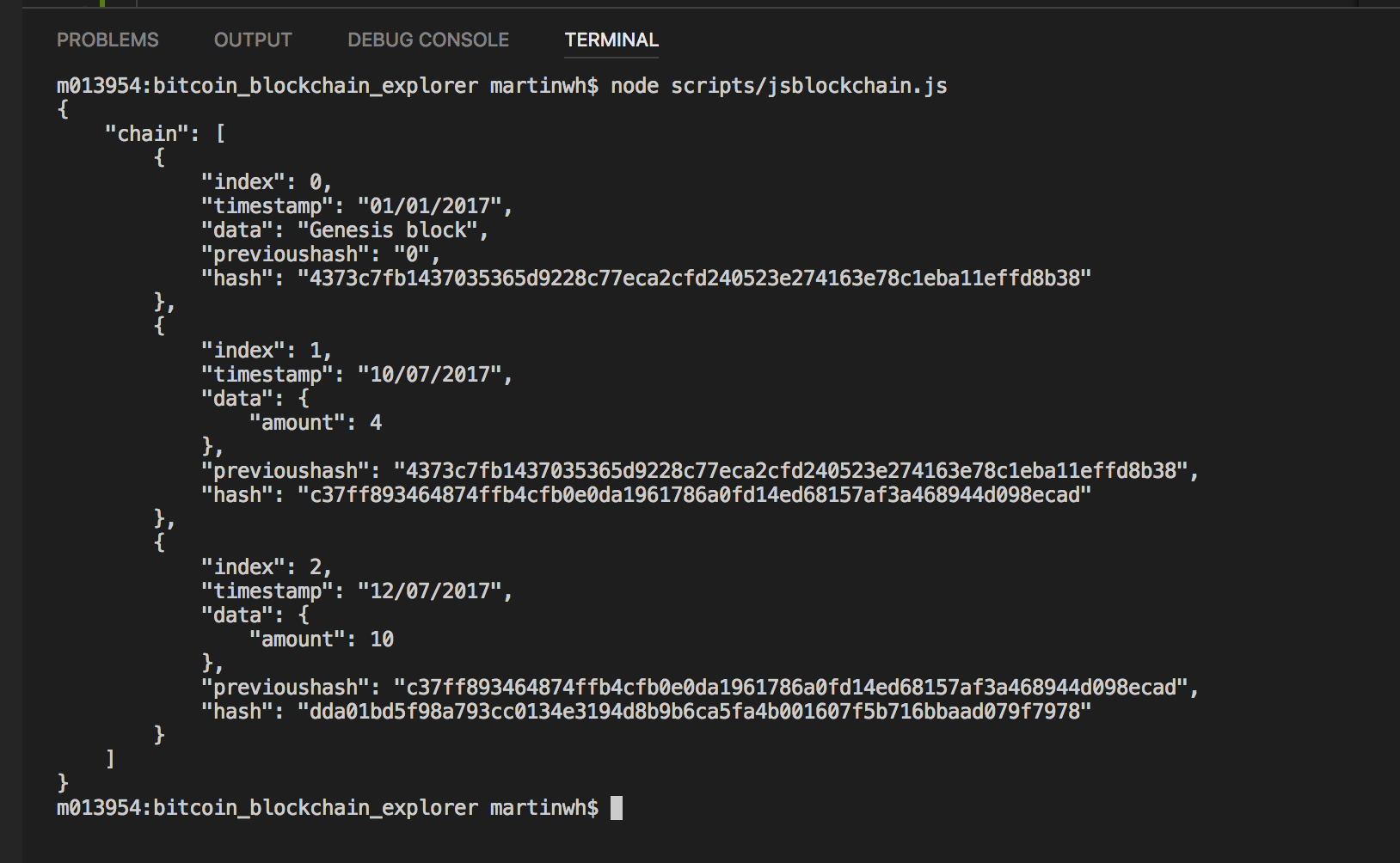


Figure 10: Running your myBitcoin blockchain

Already, at this stage, if you have completed the Blockchain Tutorial 1, you should be able to reflect on this piece of JavaScript code and consider how to re-implement it in PHP. For example, how can you invoke it from a browser application using JQuery? How can you integrate (or rebuild) it into your MVC framework, e.g. using the SQLite database and PHP to implement classes and associated methods for the blocks and the blockchain. You would also need the PHP controller methods to invoke the PHP methods associated with PHP block and blockchain classes. Part 3 of this Blockchain Tutorial 2 offers you the chance to do this.

Moving on, the whole tenet about a blockchain is that it should be immutable, i.e. any block data cannot be changed without invalidating the rest of the chain. Clearly though, at this stage, we could simply just change the data in a block and recalculate the rest of the chain. This is in effect exposes two issues:

1. We need to implement a method to verify the integrity of our blockchain, so we will do that now;
2. And, later in Part 2 of this Blockchain Tutorial 2, we will implement a ‘proof of work’ that makes it computationally too expensive change a block’s data value and then revaluate all the following blockchain hash values — make it so that you need computational power greater than 51% of all the miners in the network.
   * This turns out to be critical in order to prevent what is often called the ‘double spending problem’ in digital currencies, i.e. that of preventing Alice from sending Bob 2 bitcoins, and then trying to send her sister the same 2 bitcoins — which at today’s bitstamp Exchnage Rate of $10,901 would be nice little earner if you could get away with it!

### Create a blockchain validation method

1. So, let’s create a new method in our blockchain class to verify if the blockchain is valid, i.e. isChainValid(), Figure 11.

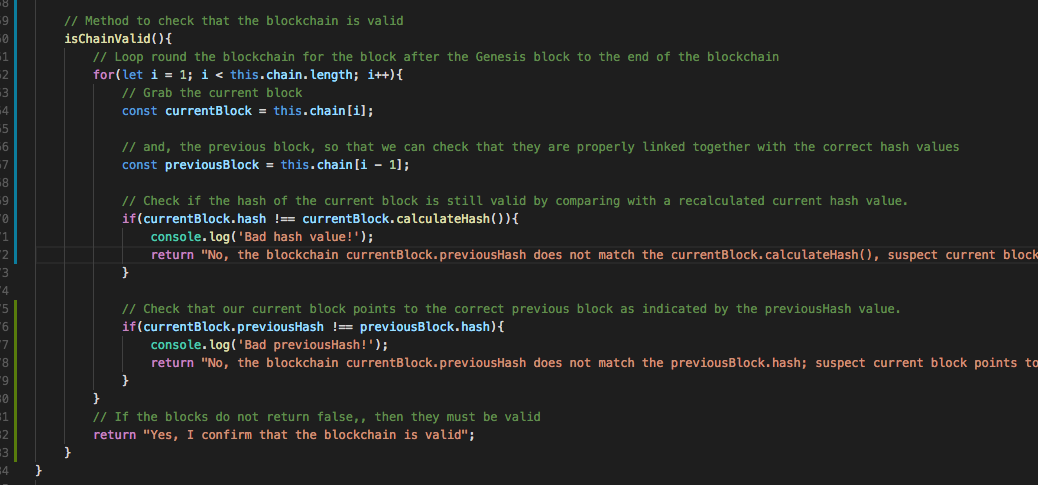


Figure 11: JavaScript methods to check the blockchain is valid

* Code explanation
  + Lines 60 -83: Implement the isChainValid() method, where first
  + Lines 62 – 80: We set up a for loop to loop around the blockchain from block 1 (we don’t need block 0 because this has no previousHash, it being the Genesis block) to the end indicated by ‘this.chain.length - 1’. And, of course, we increment the for-loop counter to traverse the chain. At each loop, we check that the current block points back to the previous block by comparing the currentBlock.previoushash value in the current block matchs the previousBlock.hash value in the previous block. If true, clearly the blocks are linked together correctly, but if false we return a message indicating so. You could just return false. I have clipped off the message in Figure 11, so don’t forget to enclose it in double quotes and end with a semi-colon. Similarly, we check that the currentBlock.hash value matches a recalculated hash value for the current block. If different, it indicates that the current block has been tampered with.
  + Line 64 - 67: we set up constants to store the current block and the previous block. We will access the current block hash and previous hash values to compare them to ensure we are linking to the current blocks in the chain and check that any data has not been tampered with.
  + Lines 70 – 73: As mentioned above, check if the hash of the current block is still valid by asking it to recalculate the current hash value. If any data value has been changed in any of the blocks the hash value (calculated with calculateHash()) will differ for the block's stored has value indicating that the data has been tampered with. This effectively makes the blockchain invalid. Of course, if the data has been changed, why not simply recalculate the hash too and store the new hash value. Of course, if the blockchain is very long, and data in an early block was changed, we might have to recalculate thousand upon thousands of blocks in the chain. But hey, we have fast computers right? This is considered in Part 2 of this Blockchain Tutorial 2. We simply make computing a hash value computationally too expensive to perform, as you'll see later.
  + Lines 76 – 79: As explained above, check that our current block points to the correct previous block as indicated by the previousHash value. So, we need to check if the previous Hash property is the correct hash set in the current block. If this is not the case, we know something is wrong because our current block does not point to the previous block; it points to some other block, perhaps a falsely created blockchain created by a dastardly cryptocurrency thief!
  + Line 82: If both checks return true, we can return true, or a simple message indicating so — the blockchain is valid

1. You will have noticed some console.log() statements. I put those in because, everyone implements bugs, so these were just to trace when we find ourselves having to debug!

### Test the integrity of our blockchain

1. Let’s test the code so far, comment out the old console.log() on line 96, and add the new console.log() statement on line 93, Figure 12.



Figure 12: Test the blockchain validation

* Code explanation
  + Lines 87 – 90: As before, this code instantiate a new blockchain and adds two blocks, so we should have three blocks as before, i.e. the genesis block and the two transaction blocks
  + Line 93: A new console.log() function concatenates a message ‘Is blockchain valid?’ with the result to be returns from the myBitcoin (the new blockchain) isChainValid() method.

1. Let’s verify the integrity of our blockchain. On the command line, you can now test, as you did before in Figure 10, by entering node jsblockchain.js, but make sure you are inside your scripts folder (or provide the path to your scripts folder), Figure 13.



Figure 13: Test the blockchain is valid

1. Ok, well, assuming no bugs, it is bound to show that the blockchain is valid because we haven’t tampered with it. Let’s do a bit of tampering, Figure 14.

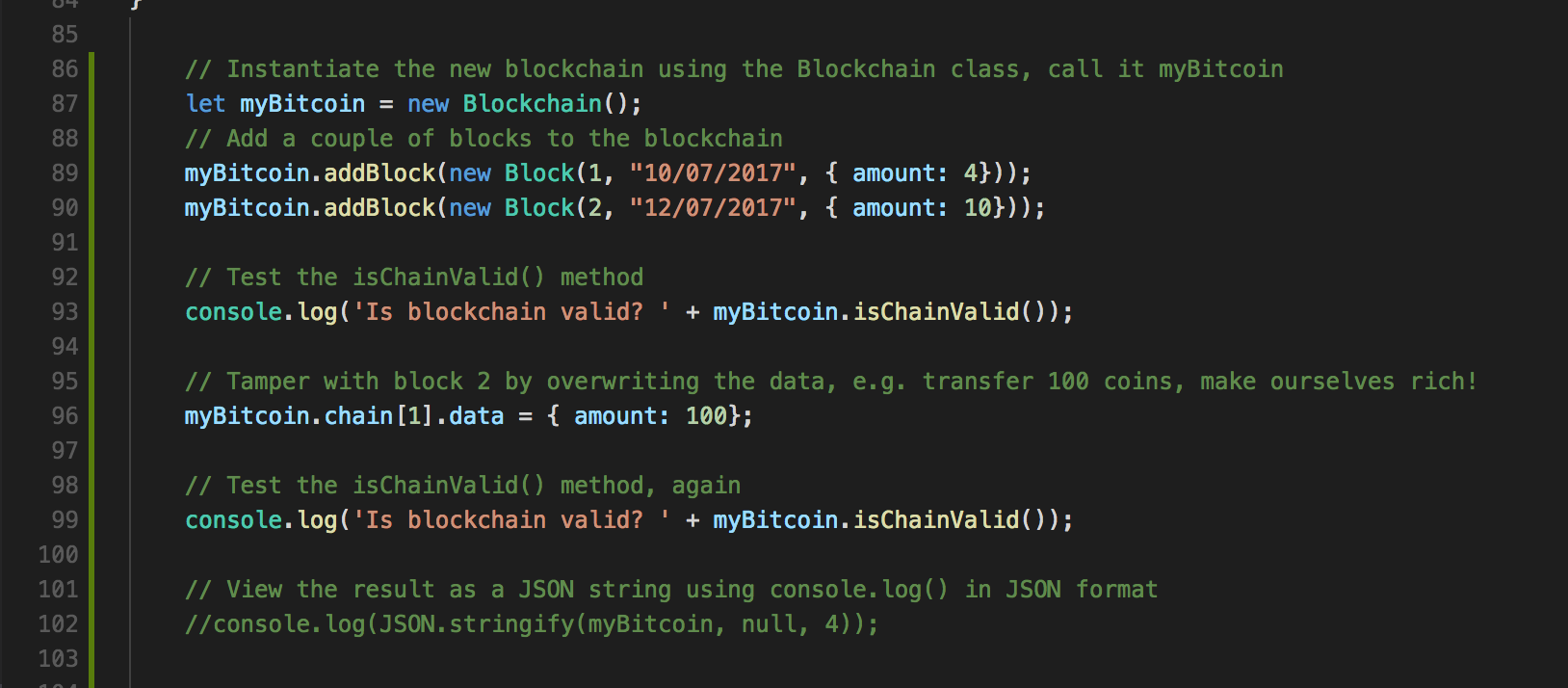


Figure 14: Tamper with the blockchain by over writing the data in block 2

* Code explanation
  + Lines 87 – 93: As before
  + Line 96: Here we are overwriting block 2 index by 1, block 1 being the zero Genesis block, by assigning (in JSON format) the amount: 100 coins.
  + Line 99: Copy the previous console.log() to validate the blockchain. So, we should see that when the blockchain is created, and the two blocks are added, it will validate ok, but when we tamper with the block it should show invalidated. Run the jsblockchain.js file again to test it, Figure 15.

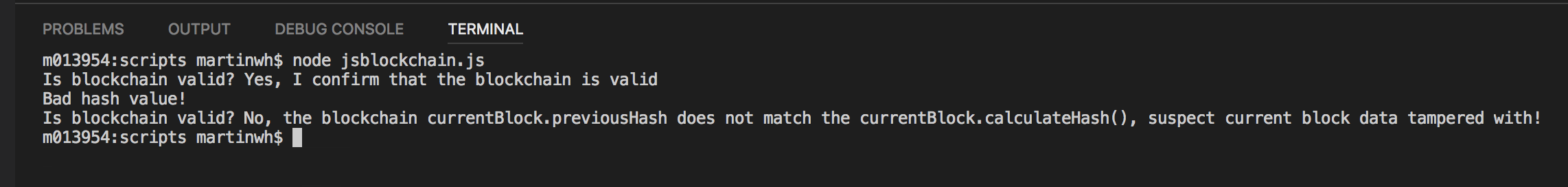


Figure 15: Test for tampering with a block’s data

1. Ok, so far, we only tampered with the data, we tried to get rich quick! Itdidn’t work, we got foiund out. But, what if we recalculate the hash value for block 2 and replace the hash value for block 2 with the new hash value. That will work, won’t it? We will then have the data, i.e. 4 coins replaced with 100 coins, and the new hash value will be correct. Let’s give it a go, Figure 16.

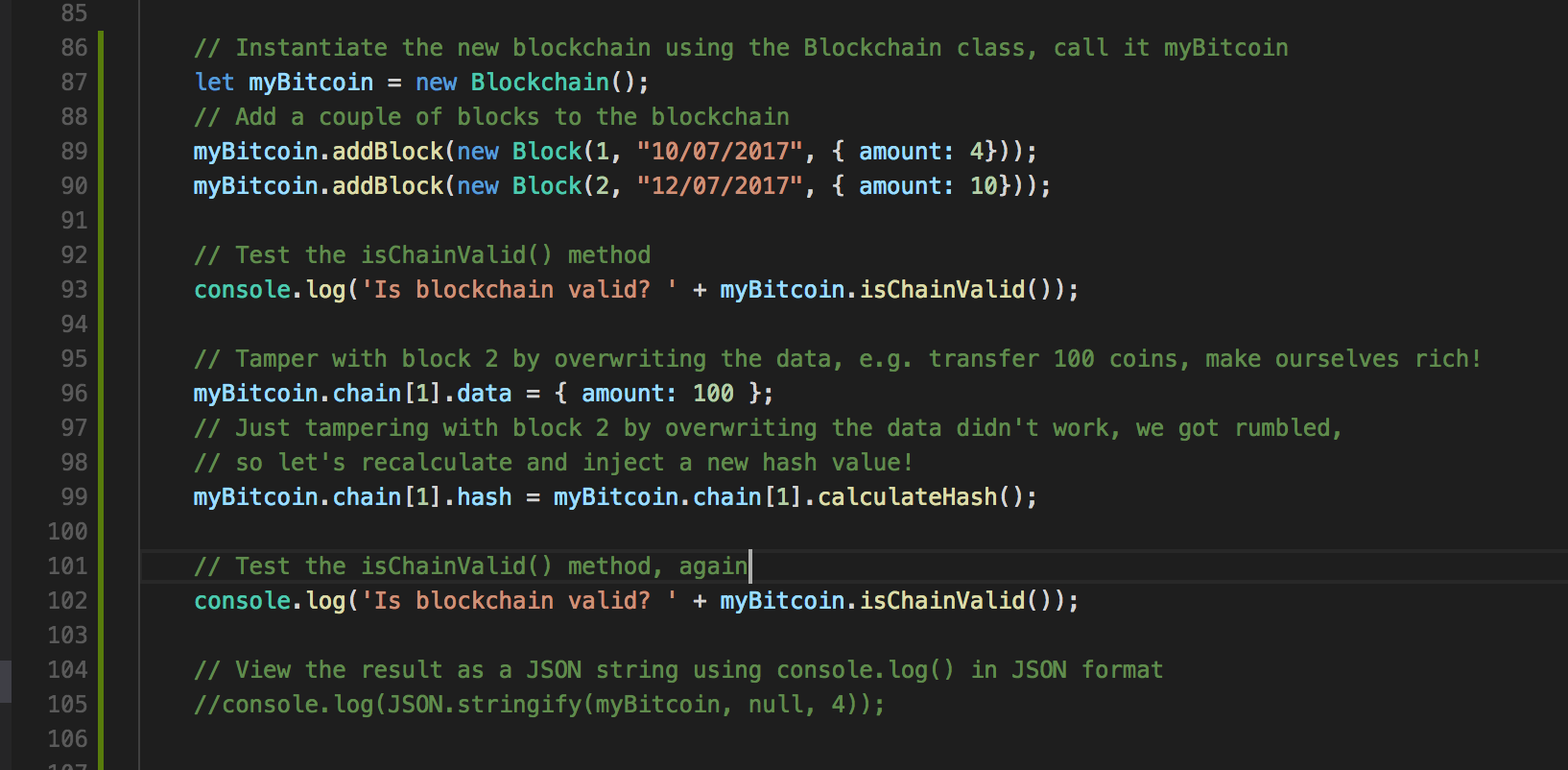


Figure 16: Let’s recalculate the hash to try and defeat the block validation

* Line 99: We are now overwriting block 2 hash value by recalculating its hash. Did this work? Figure 17 indicates that it did not. We can see that we hit our test to validate that a currentBlock.previousHash value did not match the previousBlock.hash value. Here, we see that in block 2 (index 1) the hash value starting with b5dc5… (i.e. the new recalculated hash) does not match the following block’s previousHash ea161 … Ooops! And, of course we output a message to say so.

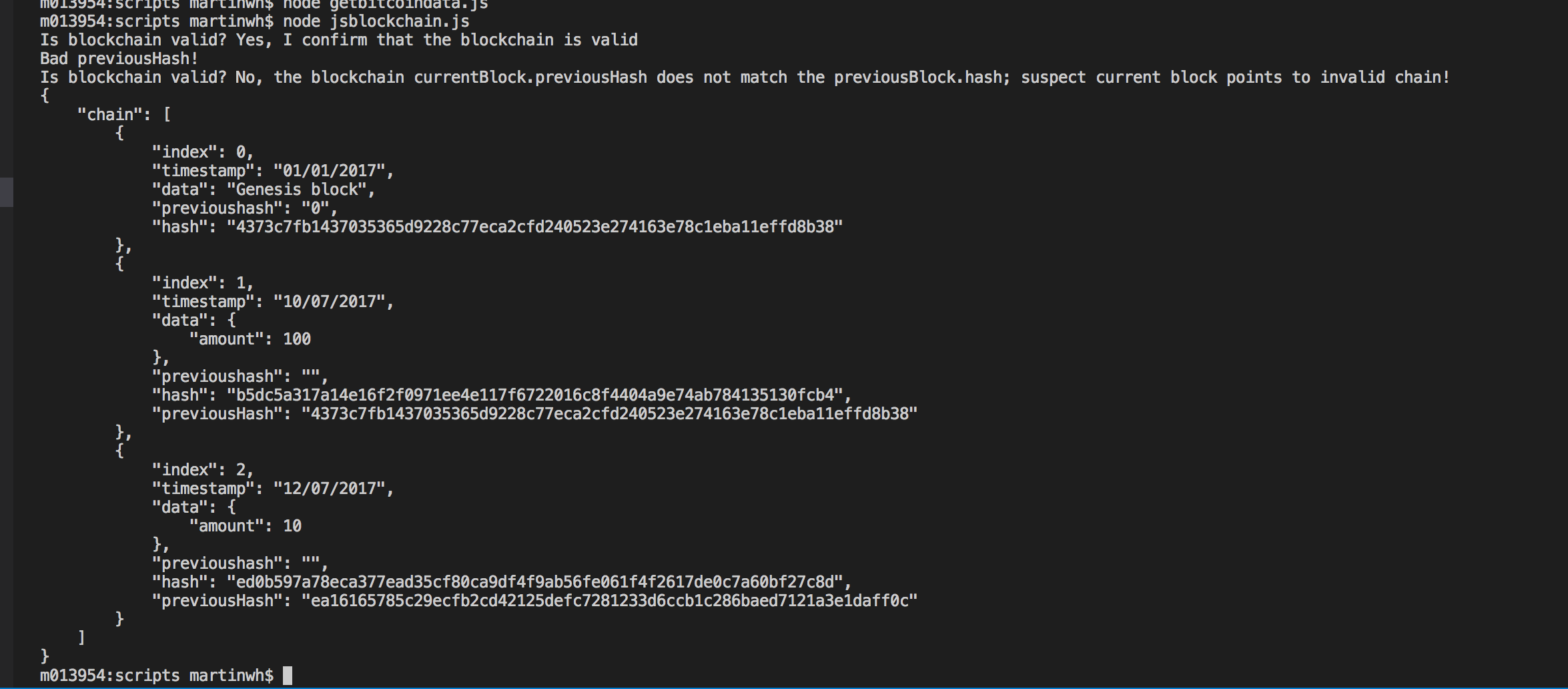


Figure 17: Test for tampering with a block’s hash value

* + We could, of course put in more code to pin point the block that has been tampered with, etc. But, suffice to say, the chain is broken somehow. We could write code to roll back the blockchain to a previous valid version, etc. but this is beyond the scope of this tutorial.
  + This simple illustration of blockchain concepts also lacks many other features, such as a ‘proof of work’, which we will look at next in Part 2 of this Blockchain Tutorial 2, a distributed, e.g. peer 2 peer, network, lots of other ‘miners’ validating the block, it also doesn’t check to see if you have enough funds (a bitcoin wallet does this), and many other limitations. However, it does demonstrate how a blockchain works.

Next, in Part 2 of this Blockchain Tutorial 2, we will look at the concept of ‘proof of work’ as a method to prevent an attacker simply recalculating hash values for the whole blockchain, thus preventing, in effect, the creation of a new valid blockchain.

## Part 2 — Implementing a ‘Proof of Work’ in JavaScript

As we are going to move forward investigating the Bitcoin blockchain, particularly when we consider exploiting APIs and WebSockets to do so, it might be instructive to consider some basic Bitcoin terminology. You have seen, in Part 1 of this Blockchain Tutorial 2, that hash functions are critical to validating a blockchain’s integrity. Hash functions are used to link blocks together in the blockchain, but they are also used for mining, generating addresses (e.g. for Bob and Alice to send and receive bitcoins to), public and private key generation, and individual transactions.

* **Blockchain** — simply put, a bitcoin blockchain is a distributed shared public ledger that tracks all bitcoin transactions, and not much else — it has no smart contract technology. In slightly more detail, a blockchain is a ledger that is an append only record system shared across a peer to peer network which uses cryptographic means to secure authenticated and verifiable transactions, and the blockchain uses a consensus (validation) protocol between all network nodes (miners) to verify transactions. Clearly, the structure of the Bitcoin blockchain is rather more sophisticated than the simple example coded above. Every user of the Bitcoin software will have a copy of the Bitcoin blockchain, so as we indicated above when we wrote our rather simple methods to verify the integrity of our blockchain, if an attacker tried to tamper with their copy of the blockchain it would generate a subsequent chain that would stick out like a sore thumb against the many thousands of valid blockchain copies, hence one would not use it.
* **Mining** — Miners compete with other miners to solve a complicated mathematical problem, i.e. a cryptographic hash function. For bitcoin mining, about every 10 minutes, a miner tries to match this pre-formatted cryptographic hash value to solve a block of transactions. If a miner solves the block first then that miner is rewarded, currently with 12.5 BTC (Bitstamp Last Price was: $9730.00 on the 11/30/2017 at 12:23:47) — a potential profitable endeavour. However, it is not easy to solve the cryptographic hash function, which is basically a one-way encryption without a key. The hash function takes an input and returns a seemly random fixed length hash value — you saw this in the Part 1 Blockchain Tutorial 2 above where we used the SHA256 cryptography library to create the JavaScript calculateHash() method. We saw that any change to the input will create a completely different has value, and it is this randomness that makes it impossible to predict the output of the hash function. We can, therefore, use this characteristic as a ‘proof of work’ for validating the block. In reality, the hash value that a miner has to match will have lots of leading zeros, the length of which is an indication of a measureable difficulty. That is, we can more or less determine how long it might take to find a match, but there is no way to find a match any better other than by guessing. This is why we have the evolution of hardware used to solve the cryptographic hash value for the block. In the beginning miners used a simple PC, then they discovered that the Graphics GPU could be programmed to do it faster, subsequently this led to the development of ASIC miners specifically developed to solve the make guesses at the hash value billions of times a second. Such arrays of hardware are not cheap to run and consume lots of electricity. That is why 80% of mining operations, for example, are located China where electricity is cheap[[1]](#footnote-1). So, in summary mining is the process of solving a cryptographic hash function to validate block transactions, and is subsequently rewarded with bitcoins
* **Block** — Blocks in the Bitcoin blockchain are basically batches of transactions that are queued waiting to be validated within a specific timeframe (every 10 minutes). We saw in our simple example above where we manually did two transactions (input to the data property), in reality hundreds, maybe more (have a look at a typical Bitcoin block) would be hashed together with the index, previoushash, timestamp, etc. and linked to the hash from the previous block, thus creating a chain, hence the term blockchain. As already discussed, and demonstrated in the Part 1 Blockchain Tutorial 2, if Alice tried to ‘double spend’ she would have to modify a transaction in that block, recalculate the hash, and recalculate hashes for all subsequent blocks, i.e. mine the altered block and all subsequent blocks. But, and this is critical, Alice would have to do this within one time frame (roughly 10 minutes) before everyone else on the network (all other miners) could mine one block. This becomes computationally infeasible when the rest of the chain is significantly longer.
* **Proof of Work** — The concept of ‘proof of work’ is already hinted at above, it simply refers to the hashing rate or computer power required to mine (match the cryptographic hash value) one block. As discussed, we can only find the matching hash value by trial and error, i.e. guessing, then the act of finding a valid block, i.e. doing all the computational work required to find the hash value is proof that the work has been done. You can think of it as a mechanism to slow down the validation of the block transactions such that if Alice tried to double spend her bitcoins, she would have to solve potentially thousands upon thousands of hash values inside the time it takes to solve one block. Alice is unlikely to have the computational power to do that. However, there is the notion of a 51% attack, I’ll leave you to read on[[2]](#footnote-2).
* **Address** — These are basically public hexadecimal strings used for receiving bitcoins. They are basically derived from a private key through the use of a hash function. The private key is used to control access to the bitcoin address, and anyone who has that private key can spend any unused outputs at that address. Similarly, if you lose your private key, then I am afraid, it is ‘tears before bedtime’, you just lost your bitcoins[[3]](#footnote-3). Many people of lost their bitcoins, check out this Telegraph article: [The £625m lost forever — the phenomenon of disappearing Bitcoins](http://www.telegraph.co.uk/technology/news/11362827/The-625m-lost-forever-the-phenomenon-of-disappearing-Bitcoins.html).
* **Private key** — A private key is basically a password, a 256-bit number. You can convert a 256-bit number composed of 0’s and 1’s into a Bitcoin private key, which through the use of hash functions, again, can be converted into an address. There are open source algorithms that allow you to create a private key, and a Bitcoin wallet will generate a unique address for each transaction for you[[4]](#footnote-4).

### Implement a ‘proof of work’ for our blockchain

So far in this tutorial we can see that we can create new blocks really quickly, all we need to do a transaction, calculate its hash, and add it to an array. Modern computers can do this incredibly quickly, but we don’t want people to create hundreds of thousands of blocks per second and spam our blockchain

Also, as we have seen, we can easily tamper with a block’s data, e.g. Alice tries to overwrite a previous block entry and recalculates the hash of this block, she can then recalculate all the hash functions for the blockchain, ending up with a new valid blockchain — even though it has been tampered with!

Clearly, we need to prevent Alice from doing this, because if it were allowed to happen confidence in the blockchain and its data, e.g. bitcoin transactions would evaporate, no one would use it. We therefore have a security problem, we must prevent Alice from double spending! The solution to this lies in the implementation of a ‘proof of work’, as described above, for our blockchain. Again, as described above a ‘proof of work’ requires that we have put in some effort (computationally) into making a new block — this process is also called mining.

Bitcoin mining requires the hash value of a block to begin with a certain number of zeros — the more zeros, the more difficult it is (computationally) to ‘guess’ the hash. Further, because you cannot influence the output of a hash function you simply have to make many guesses to try and find a hash value that matches the required hash function with the leading zeros — you have to hope that your guess matches the hash function before competing miners’ guesses. Remember, we said above, that Bitcoin can also vary the difficulty of matching the hash by increasing the required number of leading zeros in the hash value. Bitcoin uses this difficulty mechanism to keep the verification and addition of one new block to the blockchain about every 10 minutes. In practice, Bitocin increase the difficulty as more miners mine and their computing power gets better, such that the minors are making correct guesses faster, thus bitcoin increases the difficulty to produce a steady stream of blocks added to the blockchain, one every 10 minutes. Now, let’s implement some code to mine a block!

1. We will add a new method to the Block class called mineBlock() that takes as input a difficulty attribute, which will simply be a number to determine the number of leading zeros required in a matching hash value.

1. https://www.huffingtonpost.com/ameer-rosic-/what-is-bitcoin-mining-a-\_b\_13764842.html [↑](#footnote-ref-1)
2. https://www.investopedia.com/terms/1/51-attack.asp [↑](#footnote-ref-2)
3. https://bitzuma.com/posts/six-things-bitcoin-users-should-know-about-private-keys/ [↑](#footnote-ref-3)
4. https://en.bitcoin.it/wiki/Address [↑](#footnote-ref-4)